Office de la propriété intellectuelle du Canada
Un organisme d'Industrie Canada

Canadian Intellectual Property Office
An Agency of Industry Canada

PCT / CA 99/01216
16 FEBRUARY 2000 (16.02.00)
09/869047

4

CA99/1216

# Bureau canadien des brevets
## Certification

# Canadian Patent Office
## Certification

La présente atteste que les documents ci-joints, dont la liste figure ci-dessous, sont des copies authentiques des documents déposés au Bureau des brevets.

This is to certify that the documents attached hereto and identified below are true copies of the documents on file in the Patent Office.

Specification and Drawings, as originally filed, with Application for Patent Serial No: **2,256,970,** on December 23, 1998, by **TRUESPECTRA INC.,** assignee of Stephen B. Sutherland, Dale M. Wick, John-Paul J. Gignac and Sam D. Coulombe, for "Method for Accessing and Rendering an Image".



**PRIORITY DOCUMENT**
SUBMITTED OR TRANSMITTED IN COMPLIANCE WITH RULE 17.1(a) OR (b)

S. Grégoire
Agent certificateur/Certifying Officer

February 16, 2000
Date

Canada

(CIPO 68)

OPIC CIPO

## ABSTRACT OF THE DISCLOSURE

The invention provides a method of defining and rendering an image comprising a plurality of components (bitmaps, vector-based elements, text and effects or other effects) and an alpha channel. The components are grouped into a ranked hierarchy based on their position relative to each other. There can be groups of groups. With this grouping, each component can be defined using a common protocol and rendering and processing of the components can be dealt with in the same manner. The image can be processed on a scanline-by-scanline basis. For each scanline analysis, information regarding neighbouring scanlines are acquired and processed, as needed.

Vector based programs such as COREL DRAW™, produce a bitmap of the final image for the raster device. Similarly, the graphic program PHOTOSHOP™ produces a bitmap of the final image.

Vector based drawings tend to use little storage before rendering, as simple descriptions often produce largely significant results. Vector drawings are usually resolution independent and they are made up of a list of objects, described by a programming language or other symbolic representation. Bitmap images, in contrast, are a rectangular array of pixels wherein each pixel an associated color or grey level. This image has a clearly defined resolution (the size of the array). Each horizontal row of pixels of the bitmap is called a scanline. Bitmaps tend to use a great deal of storage, but they are easy to work with because they have few properties.

Recently the ability to combine layers of content has been standardized by using a so called "alpha channel" which represents the transparency of an object or pixel. There are levels of transparency between solid and transparent, which could be represented as a percentage. Although some standard file formats such as CompuServe's GIF are limited to 2 levels (solid and transparent), newer formats such as Aldus' TIFF, PNG (Portable Network Graphics) and the Digital Imaging Group's ".fpx" format allow 256 or more levels of transparency, which allows for smooth blending of layers of content. Normally manipulation of alpha channel information is limited to bitmap based programs.

There remains a need for a method which allows the compact descriptions of vector programs, with a retargetable output resolution, which additionally allows for full use of all of the powerful image processing

effects of an bitmap based program including the alpha channel capabilities. Our earlier U.S. Patent application SN 08/629,543 entitled Method Rendering an Image allows for scanline based rendering and divides all objects into a tool and region where the region acts as a local alpha channel for the tool. This doesn't allow for a more general use of alpha channels to create holes in images when used, for example, on web pages -- showing through the background. Also some types of objects such as formatted text with color highlighting cannot be represented easily with a separate region (as the shape of the text), and tool (with the coloring for the text) since particular words need to have different colors, and these need to follow the words when the text is reformatted. Additionally the interface is inconvenient to use as it returns a variable number of scanlines, including none, when the output device works best with at most and at least one scanline for each call.

## SUMMARY OF THE INVENTION

The present invention allows the user to define an image using different definitions for the individual objects of the image. The objects can be defined as a region tool and an alpha channel, as a bitmap and alpha channel or as a group of objects where each object within the group is defined as a region tool and alpha channel or a bitmap and an alpha channel. A group of objects is defined to act as any other object in the string of ranked objects defining the image. Grouped objects have the same defining characteristics as an object defined by a region tool and an alpha channel or a object defined by a bitmap and an alpha channel. A group object can contain within its grouping, a further grouped object which can also contain grouped objects. With this definition, each grouped object can be defined using the common protocol and the rendering of objects and the processing of objects can all be dealt with in the same manner. This arrangement

allows for all of the advantages of being able to group objects while having a common and consistent manner for dealing with the storage and rendering of an image defined by the different types of objects.

A method for rendering an image on a scanline by scanline basis where the image is composed of a plurality of distinct segments, according to the present invention, comprises defining each distinct segment of the image as a) a region tool and alpha channel, b) a bitmap and alpha channel, or c) a group of objects where the objects are defined according to a) and b), where each definition includes information of the scanline of the image affected by the particular segment.

The method further includes defining an order of the distinct segments from lower to higher in the unit and successively returning scanlines of the image where each scanline is returned by one examining the segments to determine which segments and the order of the segments which order the scanline to be returned, to examining the determining order segments of step 1, and determining the particular scanlines to be outputted by each ordered segment for returning the particular scanline of the image, and 3) using the ordered segments from lower to higher and returning the determined scanlines of the segment used by the next higher segment as an input until a scanline of the image is returned.

According to an aspect of the invention, the image includes a background segment which is defined as a region tool and alpha channel and the background segment is applied as a last segment prior to returning a scanline.

According to yet a further aspect of the invention, the method includes using an initial input for the lower most object equivalent to transparent scanlines.

According to yet a further aspect of the invention, the step of examining and determining the order segments of step 1 and determining the particular scanlines to be outputted by each ordered segment for returning the particular scanline of the image is carried out by examining the segments from the highest segment to the lower segment.

According to yet a further aspect of the invention, wherein some of the segments include a lookaround object which requires at least several scanlines from a lower object to return a scanline.

According to yet a further aspect of the invention, any defined group of objects can include as part thereof, a further group of objects.

An image made up of a number of render objects will have a "RenderLayer" as the base render object, which has a list of render objects contained in it. To create a RenderLayer job (which implements a render job interface), the RenderLayer first surveys the ordered list of objects from top to bottom, to determine the dependencies of the contained objects, and determines the amount of look around for each object. The RenderLayer job renders a scanline on request by creating a buffer for all active contained objects, that is, all objects that affect the current scanline. Each active object is partially rendered, in order, from bottom to top. The minimum portion of each object is rendered. That portion being the minimum number of scanlines required to create an output scanline for the parent object. Buffers that will be needed on subsequent calls to the render engine are retained for efficiency in calculations. Information buffered for objects can be reused or deleted when they are no longer needed. This provides for both computational and memory efficiency and allows for scanlines to be output sooner. A render object can combine its output using a variety of operators that

allow for special effects such as punching a hole through a background.

Each object is separately maintained in storage and has its own resolution or an unlimited resolution where possible. The rendering effect of each object is at the best resolution for imparting the rendering effect of the object to each segment of the scanline which the object affects. For example an object may only affect a middle segment of the scanline and the best resolution of the object for this segment of the scanline is used.

A method for grouping dependent elements of an image is provided where the method groups each element of the image into either (i) a region, tool and alpha channel; (ii) a bit-map and alpha channel, or (iii) a group of objects defined according to (i) or (ii), where each group includes information of the scanlines of the image affected by a particular element and creating a plurality of single dependencies between each subordinate element and its parent.

## BRIEF DESCRIPTION OF THE DRAWINGS

Preferred embodiments of the invention are shown in the drawings, wherein:

Figure 1 displays the abstract interfaces for the render object and render job;

Figure 2 defines the support classes which store information used to communicate between various parts of the method;

Figures 3, 3a, 3b, 3c, 3d, 3e, 3f, 3g, 3h, 3i, 3j, 3k, and 31 contain pseudo code illustrating the steps involved in rendering an image to an output device;

Figure 4 displays a rendering pipeline with a RenderLayer which contains two render objects;

Figure 5 displays a rendering pipeline with an effect which contains a region and a tool;

Figure 6 displays a depiction of a rendered image containing the objects defined in figure 4 and 5;

Figure 7 displays the inter-relationship of the various major classes used to define the method;

Figure 8 follows the partial rendering process of objects contained in a RenderLayer;

Figure 9 is a visual representation of the objects referred to in Figure 8;

Figure 10 is an example of how RenderLayers work to create a result shown as a list within a list;

Figure 11 is a hierarchical representation of the objects in Figure 10; and

Figure 12 shows

(a) the a rendered final composition of the objects, with the Layer1 object rendered into the scene, from the objects described in Figure 10;

(b) the Text1 object;

(c) the Text1 object with the Shadow1 object rendered over top; and

(d) the Text1 object with the Shadow1 object rendered over top, and then the Wave1 object rendered on top of that.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

An image is defined using a common standard and according to at least three classifications. The common standard is that each classification includes a bound box definition which defines the area which that particular objects affects. A lookaround distance which is any additional information that might be required for the object to render itself as well as the alpha channel.

The first classification is a simple object defined by a region, tool and alpha channel similar to a vector program. The second classification is a bitmap object as commonly defined by a draw program. The third classification is a grouped object which is defined by a

plurality of objects which can include further grouped objects. The common definition or standard allows grouped objects to be rendered in a similar manner to the series of objects defining the image. It also simplifies the calculations necessary for rendering a scanline as a group of objects as a common definition which allows the requirements of that particular group to be known to the other objects within a series of objects. As far as the adjacent higher and lower object is concerned, a grouped object is merely a different type of object having common characteristics, and as such, the higher and lower objects continue to interact with the objects in a common set manner.

This standard for defining an image makes it convenient during rendering of the object to look from the top down through the primary objects to determine the number of lines required of the lower objects for passing onto an upper object. This process is essentially repeated for any grouped object. In this way, the steps and interfaces for rendering of the image are consistent and straightforward.

Figure 1 shows the following structures used by the invention.

1. Abstract RenderObject Class

This class defines the minimum interfaces that an object needs to be used by the method. They are getBoundBox which returns an upright rectangle which defines the limits of the area which the object affects, getLookAround which defines the amount of extra information required around any given pixel, in order for that pixel to be rendered correctly, and initRender which returns a RenderJob object.

2. Concrete RenderLayer Subclass

The pseudo code required to implement a render object, which contains an ordered list of objects, is shown in Figure 3. A sample instance is shown in Figure 6 with a data flow diagram shown in Figure 4.

3. Other Concrete Subclasses of RenderObject

Other subclasses include Effect which contains a Region and Tool as shown in Figure 7. A corresponding data flow diagram is shown in Figure 5. Other subclasses include other effects such as Rich Text with Color Highlighting or Alpha Channel Bitmap.

4. Abstract RenderJob Class

The class defining the interfaces for a RenderJob is shown in Figure 1. They are prefersToOverwriteInputBuffer which facilitates negotiation of whether the output buffer is different from the input buffer for computational efficiency. The getLookAroundDistances is the same as in the RenderObject. Finally renderNext takes an input buffer, and an output buffer, and renders one scanline.

5. A Concrete Subclass of RenderJob for Each Concrete Subclass of RenderObject, Including RenderLayer

The pseudo code required to implement a render job for a RenderLayer is shown in Figure 3. The definitions used in figure 3 are simplified for readability. A sample walk through of how this method renders 3 scanlines is included in Figures 8 and 9.

The defining implementations shown in Figure 2 of the invention include RGBAScanline, Rectangle2D, Affine2D, LookAroundDistances. Although the implementation in Figure 3 uses an RGB color space, this method applies equally well to other color spaces such as CIE-LAB, CMYK, XYZ, etc.

Figure 9 shows an example of the operation of the invention, wherein:

a composition of 3 render objects contained in a RenderLayer is shown. The objects are "Heart1" which displays a red heart, "Hello1" which displays the black text "Hello World", and "Blur1" which blurs or makes fuzzy the content underneath it. Heart1 and Hello1 are known as "simple render objects" because they require only the background immediately behind the output pixel. This information can be ascertained by using the getLookAroundDistances method on the given object. This call passes in the output resolution and a transformation to the output space (which can involve rotation, scaling, translation and skewing - ie. all affine transformations). The result is the number of extra pixels required as input which are above, below, to the left and to the right of any given pixel, in order for the object to be rendered. When the number of extra pixels is 0 in every direction, the object is considered to be a simple object. If the number is greater than zero in any direction then the object is a "look around" object. An example of a look around object is Blur1. Blur1 requires an extra pixel in each direction to render its effect. The extra area required by the blur is shown by the dashed line around the blur's bound box in Figure 9. Note that the blur requires information below the third scanline, which means that an additional scanline which isn't output needs to be at least partially rendered.

Using a technique known as called the Painter's Algorithm, a buffer large enough to buffer the entire image is allocated. First, the background is filled in (Figure 8i steps 1-4), then the bottom most object, Heart1, is rendered completely (Figure 8i steps 5-7), next the object in front of Heart1. (Hello1) is rendered completely (Figure 8i steps 8-9) and finally, the front most object, Blur1, is rendered completely (Figure 8i steps 10-11) using the results of steps 6, 7, 8, 9. Once this

process is complete the 3 requested scanlines can be output (Figure 8i steps 12-14).

To start using the reordered rendering method, the render engine is invoked on the containing RenderLayer, called "RenderLayer1." RenderLayer1 returns a render job object identified here as "RenderLayerJob1." To get a scanline, the renderScanline method is called on RenderLayerJob1, passing in a background. RenderLayerJob1 determines which objects affect the Scanline 1 and renders them completely (Figure 8ii steps 1 and 2). The result of Figure 8ii step 2 is needed by the blur, which is buffered for later use. The resulting Scanline 1 is then returned in Figure 8ii step 3. The next time renderScanline is called, the blur becomes active. Since the blur needs a pixel above and a pixel below it in order to render correctly, the RenderLayerJob1 must buffer up more information. The result of Figure 8ii step 4-6 is buffered as well as the result of step 7-8. These three results (from step 2, 6 and 8) are then passed into the BlurJob1 which results in step 9. The buffer from step 2 can now be discarded or marked for reuse. The resulting scanline 2 is returned in step 10. To rendered scanline 3, the blur requires more than the already buffered result of step 6 and step 8, and so RenderLayerJob1 renders step 11 and step 12. These three buffers (from step 6, 8 and 12) are then passed into the BlurJob1 which results in step 13. Finally the scanline 3 is returned in step 14, and all of the temporary buffers can be discarded.

In this example, 3 scanline buffers were required versus 4 scanlines buffers with the Painter's Algorithm. With a larger render, the resource savings are often significant. Also the result of the top of the image became available much earlier.

Figures 4, 5 and 6 show an example of the processing of images by the modules in the invention.

Figure 6 shows an image of a heart ("Heart1") in its bound box beneath the text Hello World ("Hello1") , in its bound box. Both the Heart1 and the Hello1 have colour and alpha-channel attributes "(c,a)". The composite image is referred to as "RenderLayer1".

Figure 4 illustrates the processing of the entire image. First, the Background color and alpha channel information (c,a) is fed to the RenderLayer1 module, which initiates RenderJob. Starting from the bottom element, Heart1, a transparent background is fed to the subordinate call of RenderJob for Heart1. After the subordinate call of RenderJob for Heart1 has completed its processing, it returns colour and alpha-channel attributes to the calling RenderJob for RenderLayer1. These returned attributes are forwarded to the next subordinate call of RenderJob, i.e. the call relating to Hello1. Once its processing is completed, its results are returned to RenderJob for RenderLayer1. At that point, RenderJob takes the final color and attribute information from RenderJob for Hello1 and combines it with the background colour input to produce the final output color and alpha information.

Figure 5 illustrates subroutine calls within RenderJob for Heart1. Here the background color and alpha-channel information is fed to the RenderJob for the Shape of Heart1. The RenderJob for Shape returns alpha information to RenderJob for Heart1. This information along with the initial color information is fed to the ToolJob module for the Solid Color of Heart1. This module returns colour and alpha-channel attributes to the calling RenderJob for Heart1. These returned attributes are forwarded to the next subordinate call of RenderJob, i.e. the call relating to Hello1.

In another example, Figure 12 shows a composite image comprising a heart, stylized text "Exploring the Wilderness" and a bitmap image of an outdoor scene

underneath the heart and the stylized text. The stylized text is shown with its normal attributes at 12b, with a shadow at 12c and with a wave at 12d.

As shown in figure 10, the invention processes each element of the image according to a hierarchical stack, having the heart ("Heart1") at the top of the stack, the stylized text ("Layer1") in the next layer down and finally with the bitmap ("Bitmap1") at the bottom. Layer1 is exploded to show its constituent effects, comprising a wave effect ("Wave1"), a shadow effect ("Shadow1") and the text ("Text1").

Figure 11 shows the hierarchy structure of the image, where the RootLayer is the fundamental node, representing the image. Elements of the image, i.e. Heart1, Layer1 and Bitmap1 are shown as immediate dependents of the RootLayer. Further sub-dependencies of Layer1, i.e. Wave1, Shadow1 and Text1 stem from Layer1. Other information, such as the bound box region may also be associated with each element. It can be appreciated that this structure of the invention isolates the dependencies between parent and child elements to one level of abstraction. As such, the invention provides abstraction between and amongst elements in an image. This abstraction provides implementation efficiencies in code re-use and maintenance. It can be appreciated that for more complex images having many more elements, bitmaps and effects, the flexibility and efficiencies of using the same code components to processes the components of the image become more apparent.

In the preferred embodiment, exactly one scanline is rendered during each call to the render method on any render object. This even holds for render groups, since Render Layer constitutes a valid implementation of the Render Object class. In the example implementation, render group always passes a completely transparent

background as input to its bottommost object. Then the scanlines produced by applying the bottom most object to the transparent background scanlines are passed as input to the next higher object. Similarly, the output of the second object is passed as input to the third object from the bottom. This passing repeats until the cumulative effect of all of the render group's objects is produced. This final results are then composited onto the background scanlines (passed by the caller) using the render group's compositing operator.

Because some render objects have forward look-around, it is often necessary for lower objects to render a few scanlines ahead of objects above them. For example, for an object with one scanline of forward look-around to render a single scanline within its active range, the object immediately below it must already have rendered its result both on that scanline and on the following scanline. Since rendering must be performed from the bottommost object to the topmost object, therefore, in order to guarantee that a single scanline will be completely rendered by all objects by the end of a call to the rendering method, it is useful to begin the process by determining exactly how many scanlines must be rendered by each object in the render group.

The computation is most easily done in terms of the total number of scanlines rendered by each object so far during the entire rendering process, as opposed to the number of scanlines rendered by each object just during this pass. The total number of scanlines required _of_ an object is referred to, relative to that object, as downTo whereas the total number of scanlines required _by_ an object is referred to, relative to that object, as downToNeeded. Note that the downToNeeded of a given object is always equal to the downTo of the object immediately below it, if applicable. In the case of the bottommost object, its downToNeeded is the number of empty input

scanlines that must be passed to it in order for it to satisfy the object above it, if any, or the caller otherwise.

Although various preferred embodiments of the present invention have been described herein in detail, it will be appreciated by those skilled in the art, that variations may be made thereto without departing from the spirit of the invention or the scope of the appended claims.

THE EMBODIMENTS OF THE INVENTION IN WHICH AN EXCLUSIVE PROPERTY OR PRIVILEGE IS CLAIMED ARE DEFINED AS FOLLOWS:

1. A method of rendering an image on a scanline by scanline basis where the image is composed of a plurality of distinct segment, said method comprising
defining each distinct segments of the image as
a) a region, tool and alpha channel,
b) a bit map and alpha channel, or
c) a group of objects where the objects are defined according to a) and b) and where each definition includes information of the scanlines of the image affected by the particular segment;
defining an order of the distinct segments from lower to higher in the image, successively returning scanlines of the image where each scanline is returned by
i) examining the segments to determine which segments and the order of the segments which affect the scanline to be returned,
ii) examining the determined ordered segments of step I) and determining the particular scanlines to be outputted by each ordered segment for returning the particular scanline of the image, and
iii) using the ordered segments from lower to higher and returning the determined scanlines of the segment used by the next higher segment as an input until a scanline of the image is returned.

2. A method as claimed in claim 1 wherein said image includes a background segment defined as a region, tool and alpha channel and said background segment is applied as a last segment prior to returning a scanline.

3. A method as claimed in claim 2 wherein said method includes using an initial input for the lower most object equivalent to transparent scanlines.

4. A method as claimed in claim 3 wherein a group of objects is a simple object which requires only one scanline of input for returning a scanline of output.

5. A method as claimed in claim 1 wherein said step of examining the determined ordered segments of step I) and determining the particular scanlines to be outputted by each ordered segment for returning the particular scanline of the image is carried out by examining the segments from the highest segment to the lowest segment.

6. A method as claimed in claim 1 wherein at least some of said segments include a look around object which requires at least several scanlines from a lower object to return a scanline.

7. A method as claimed in claim 6 wherein said group of objects contains at least 3 objects.

8. A method as claimed in claim 1 wherein a group of objects includes as part thereof, a further group of objects.

9. A method of grouping dependent elements of an image for processing on a scanline by scanline basis, said method comprising: grouping each element of the image as

a) a region, tool and alpha channel;

b) a bit map and alpha channel; or

c) a group of objects where the objects are defined according to a) and b) and where each group includes information of the scanlines of the image affected by the particular element; creating single depending associations between each subordinate element and its parent; and

defining an order of the distinct elements from lower to higher in the image.

Figure 1.

Abstract Interfaces for Hierarchical Model

```
Interface RenderObject
          Rect2D getBoundBox(Affine2D objToHome)
          LookAroundDistances getLookAround(int width, int
height, Affine2D objToHome)
          RenderJob initRender(int width, int height,
Affine2D objToHome)
EndInterface
```

```
Interface RenderJob
          boolean prefersToOverwriteInputScanline()
          LookAroundDistances getLookAround(int width, int
height,Affine2D objToHome)
          void renderScanline(RGBAScanline inputBuffer,
RGBAScanline outputBuffer)
EndInterface
```

Figure 2.

Support Classes.

```
Class RGBAScanline
          Constructor(int length, LookAroundDistances
look)
     Byte[] getRGBData(int scanline)
     Byte[] getAlpha(int scanline, int offset)
     Int getRGBOffset(int scanline)
     Int getAlphaOffset(int scanline)
EndClass

Class LookAroundDistances
          Int left,right,up,down
EndClass

Class Rectangle2D
          Double x,y,width,height
EndClass

Class Affine2D
          [ a11 a12 a13]
          [ a21 a22 a23]
          [   0     0     1 ]
EndClass
```

Figure 3a.

RenderLayer and RenderLayerJob Psudocode

This algorithm is presented here in simplified form in order to clearly convey its fundamental concepts. All trivial modifications or extensions to the algorithm as presented here should be considered as such, even if the modifications are only conceptually trivial, though complex in implementation.

The % operator is defined as follows: a % b := b*(a/b-floor(a/b))

```
-----------------------------------------------------------------------
-

// Each instance of this class represents an active render process
// for a render object.  Rendering begins by calling
// RenderObject.initRender(), which returns a render job.
abstract class RenderJob
{
        // Renders a single scanline of an object onto the given
        // scanline, storing the result in the given output buffer.
            It
        // is acceptable for out to actually be the central scanline
            of
        // in.  The caller is responsible for making sure that in
        // contains a sufficient amount of look-around information.
        void renderScanline( LookAroundScanline in, PaddedScanline
out);
}

// This class represents a 2D object which has some appearance or
// effect when rendered onto a background.
abstract class RenderObject
{
        // Returns the maximum required look-around for rendering the
given
        // rectangle.
        LookAround getLookAround( Rectangle r);

        // This method should return true if this object's render
jobs
        // would prefer to write to the same buffer from which they
read.
        // This allows for a simple but effective optimization.
        boolean prefersToOverwriteInputScanline();

        // Returns a 'coverage' object loosely describing the maximum
        // extent of this object's visible effect, regardless of the
        // appearance of the background.  See also: Coverage.
        Coverage getDomain();

        // Constructs a new render job for this object which will
render
        // all of the pixels in the given rectangle.
        RenderJob initRender( Rectangle r);
}
```

Figure 3b

```
abstract class CompositingOperator
{
        // Composits the two given colours, and returns the resulting
        // colour. Note that the term `colour' is used loosely here,
        // and includes an alpha component.
        RGBA composit( RGBA background, RGBA foreground);
}

abstract class Coverage
{
        // Returns a rectangle that completely contains the area
        // represented by this object.
        Rectangle getBoundbox();
}

// This class represents a horizontal line of RGBA values, as a sub-
// array of an array of RGBA values.  This is a valuable alternative
to
// just a plain array, since it allows for look-around pixels and
// sub-scanlines.
class PaddedScanline
{
        private int offset;
        private int width;
        private RGBA[] buffer;

        // Constructs a padded scanline with a specified amount of
        // padding to the left and right.
        PaddedScanline( int width, int lookLeft, int lookRight)
        {
                offset = lookLeft;
                this.width = width;
                buffer = new RGBA[ width + lookLeft + lookRight];
        }

        // Construct a sub-scanline of the given scanline.
        private PaddedScanline( PaddedScanline s, int offset,
            int width)
        {
                this.offset = s.offset + offset;
                this.width = width;
                buffer = s.buffer;
        }

        // Returns a sub-scanline of this scanline.
        PaddedScanline subScanline( int offset, int width)
        {
                return new PaddedScanline( this, offset, width);
        }
```

Figure 3c

```
        // Gets the colour and alpha of the pixel at a specified
offset
        // from the left edge of this scanline, not counting any
        // additional padding.
        RGBA getPixel( int x)
        {
                return buffer[ x + offset];
        }

        // Sets the colour and alpha of the pixel at a specified
offset
        // from the left edge of this scanline, not counting any
        // additional padding.
        void setPixel( int x, RGBA colour)
        {
                buffer[ x + offset] = colour;
        }

        // Copies the contents of this scanline, plus additional
        // look-around information to the left and right, into the
        // given scanline.
        void blitTo( PaddedScanline dest, int lookLeft, int
lookRight)
        {
                // If the source and dest scanlines are the same
                // scanline, then we can just return without doing
                // anything.
                if( this == dest) return;

                for x = -lookLeft to (width+lookRight-1)...
                        dest.buffer[ x + dest.offset] = buffer[ x +
                        offset];
        }

        // Composits the contents of this scanline, plus additional
        // look-around information to the left and right, into the
        // given scanline. The given compositing operator is used to
        // perform the compositing.
        void mixTo( PaddedScanline dest, CompositingOperator
operator,
                int lookLeft, int lookRight)
        {
                for x = -lookLeft to (width+lookRight-1)...
                {
                        dest.buffer[ x + dest.offset] =
                              operator.composit(
                              dest.buffer[ x + dest.offset],
                              buffer[ x + offset]);
                }
        }
}
```

Figure 3d

```
// Represents a scanline with additional look-around in all four
// directions.
class LookAroundScanline
{
        private int offset;
        private PaddedScanline[] buffer;

        // Constructs a look-around scanline with the specified
amount
        // of vertical look-around.  This constructor does not
      allocate
        // the scanlines themselves.  Therefore, the object will not
      be
        // complete until all of the scanlines have been set using
      the
        // setScanline() method.
        LookAroundScanline( int lookUp, int lookDown)
        {
                buffer = new PaddedScanline[ 1 + lookUp + lookDown];
                offset = lookUp;
        }

        // Returns a specified padded scanline.  A y value of 0
returns
        // the central scanline.  Negative values return higher
        // scanlines.  Positive values return lower scanlines.
        PaddedScanline getScanline( int y)
        {
                return buffer[ offset + y];
        }

        // Sets the scanline at the given y position.  0 is the
central
        // scanline.  Negative values may be used to set higher
        // scanlines.
        void setScanline( int y, PaddedScanline s)
        {
                buffer[ offset + y] = s;
        }

        // Returns the colour and alpha values of the pixel at the
        // given coordinates in this scanline.
        RGBA getPixel( int x, int y)
        {
                return getScanline( y).getPixel( x);
        }

        // Sets the colour and alpha values of the pixel at the given
        // coordinates in this scanline.
        void setPixel( int x, int y, RGBA colour)
        {
                getScanline( y).setPixel( s, colour);
        }
}
```

WH-10,340CA

Figure 3e

```
// This is a special render object that contains a list of render
// objects.  The effect of rendering this object onto a given
// background is the same as rendering each of its contained objects
// one by one onto a blank background, then compositing the resulting
// image onto the given background.
class RenderLayer extends RenderObject
{
        Vector objects;
        CompositingOperator op;

        LookAround getLookAround( Rectangle r)
        {
                return {0,0,0,0};
        }

        boolean prefersToOverwriteInputScanline()
        {
                return true;
        }

        // Returns the union of the domains of the contained objects.
        Coverage getDomain()
        {
                Coverage r = empty;

                for each contained object...
                        r = Union( r, object.getDomain());

                return r;
        }

        // Initializes the render job.
        RenderJob initRender( Rectangle r)
        {
                return new RenderLayerJob( this, r)
        }
}
```

Figure 3f

```
// For use by RenderLayerJob, this class is essentially a wrapper
// around RenderJob which avoids initialization until it is
absolutely
// necesary, then automatically goes away when it is done.  It also
// collaborates with the RenderLayerJob in order to share buffers
// effectively with other job nodes.
class JobNode
{
        RenderObject object;
        Rectangle rect;
        LookAround look;
        int nextOut;
        RenderJob job;
        PaddedScanline[] backBufs;
        private LookAroundScanline rgbaScanline;
        int downTo;

        JobNode( RenderObject object, Rectangle r)
        {
                this.object = object;
                rect = object.getDomain().getBoundbox();
                look = object.getLookAround( r);
                nextOut = rect.top - look.up;
        }

        // Computes how many background scanlines must be passed
before
        // a total of downTo scanlines.
        int downToNeeded()
        {
                if( downTo < rect.top)
                {
                        return downTo;
                }

                if( downTo < rect.bottom)
                {
                        return downTo + look.down;
                }

                if( downTo < rect.bottom + look.down)
                {
                        return rect.bottom + look.down;
                }

                return downTo;
        }
```

Figure 3g

```
// Renders a single scanline of this object.  For efficiency,
// rendering is restricted both horizontally and vertically
// to
// pixels contained in this object's affected area.
// Vertically, this restriction is accomplished by deferring
// initialization of the render job until the object's first
// scanline is reached, then by throwing away the render job
// when its last scanline is reached.  Horizontally, the
// restriction is accomplished by passing sub-scanlines of
// the
// input buffers to the render job.
boolean renderScanline()
{
        // If we haven't yet reached scanline rect.top-
        // look.up
        // then there is nothing to do yet.
        if( nextOut < rect.top - look.up) return false;

        // Once we reach the scanline a few scanlines above
        // this object's bound box, we need to begin
         // buffering
        // background scanlines.
        // This happens look.up scanlines above the object.
        if( nextOut == rect.top - look.up)
        {
                // Create the back buffers for look-around
                // objects
                // Note that non-look-around objects that
                // don't
                // overwrite their input buffer also need a
                // back buffer.

                if( look.top > 0 ||
                      !job.prefersToOverwriteInputScanline())
                {
                        backBufs = new PaddedScanline[
                            look.up + 1];

                        for i = 0 to look.up...
                                backBufs[i] = new
                                  PaddedScanline(
                                  rect.width, look.left,
                                  look.right);
                }
        }
```

WH-10,340CA

Figure 3h

```
// If we've reached the object's first scanline,
// construct its render job.
if( nextOut == rect.top)
{
        job = object.initRender( rect);

        // We also need to construct a look-around
        // scanline as
        // input to the renderScanline() method.
        rgbaScanline = new LookAroundScanline(
              rect.width,
              look.up, look.down);
}

// The global buffers are where the final results are
// built up. The following line decides which global
// buffer this object's current scanline will be
// rendered into.  At this point, this same global
// buffer is guaranteed to contain the appropriate
// final results of all underlying objects.

// Compute which global buffer represents the current
// scanline.
PaddedScanline globalBuf = globalBufs[ nextOut %
  numGlobalBufs].subScanline( rect.left);

// If we have not yet reached the object, all we need
// to do is buffer the current scanline for next
// pass.
if( nextOut < rect.top)
{
        globalBuf.blitTo( backBufs[ nextOut %
              (look.up + 1)],
              look.left, look.right);

        nextOut ++;

        return false;
}
```

Figure 3i

```
// Set the look-down information to the contents of
//  the
// appropriate global buffers.  Note that for look-
// down
// information, we are guaranteed that the contents
 // of
// the global buffers has not yet been overidden.  /
// This
// is not true for look-up information.  This is why
 // we
// need to keep back-buffers, but not front-buffers.
for i = 0 to look.down...
{
        rgbaScanline.setScanline( i,
                globalBufs[ (nextOut + i) %
                    numGlobalBufs].
                    subScanline( rect.left));
}

// Compute the back buffer corresponding to the
// current
// scanline.
PaddedScanline backBuf = backBufs[ nextOut %
  (look.up + 1)];

// If this job prefers not to overwrite its input
// scanline, then let's give it the current back
// buffer
// as input. (The current back buffer will shortly
// become a copy of the current global buffer.)
if( !job.prefersToOverwriteInputScanline)
{
        rgbaScanline.setScanline( 0, backBuf);
}

if( look.up > 0 ||
  !job.prefersToOverwriteInputScanline())
{
        // Make a local copy of the input scanline
        // that
        // we're about to overwrite
        globalBuf.blitTo( backBuf, look.left,
        // look.right);
```

Figure 3j

```
                        // Set all of the look-up scanlines to
                        // previously stored back buffers.
                        int backBufNum = nextOut;
                        for i = -look.up to -1...
                        {
                                backBufNum ++;
                                rgbaScanline.setScanline( i,
                                        backBufs[ backBufNum %
                                        (look.up + 1)]);
                        }
                }

                // Render the scanline
                job.renderScanline( rgbaScanline, globalBuf);

                nextOut ++;

                // Return true if and only if we just rendered the
                // object's final scanline.
                return (nextOut == rect.top + rect.height);
        }
}

PaddedScanline[] globalBufs;
int numGlobalBufs;

// This is the implementation of RenderJob for render groups.
class RenderLayerJob extends RenderJob
{
        NodeList list;
        Rectangle rect;
        LookAround look;
        PaddedScanline tmpScanline1;
        PaddedScanline tmpScanline2;
```

Figure 3k

```
RenderLayerJob( RenderLayer l, Rectangle r)
{
        rect = r;

        int maxLookDown = 0;

        // Construct a job node for each object being
        // rendered.
        Rectangle clipRect = r;
        for each object in l.objects from last to first...
        {
                JobNode node = new JobNode( object, r);

                // Compute the object boundbox, expanded by
                // the
                // look-around distances.
                Rectangle eRect;
                eRect.left = node.rect.left - node.look.left;
                eRect.top = node.rect.top - node.look.up;
                eRect.right = node.rect.right +
                      node.look.right;
                eRect.bottom = node.rect.bottom +
                      node.look.down;

                clipRect = Union( clipRect, eRect);

                maxLookDown += node.look.down;

                list.add( node);
        }

        // Compute the required number of global buffers.
        // This
        // calculation could be improved significantly.  This
        // number must be at least 1 + max( look0,
    maxLookDown)
        // where look0 is the total vertical look-around at
        // scanline 0, and maxLookDown is the maximum
        // downwards look-around for any scanline of the
        // image.
        // It is easy to show that the number computed here
        // is
        // sufficient.
        numGlobalBufs = 1 + (rect.top - clipRect.top) +
          maxLookDown;

        // Allocate the global buffers.
        globalBufs = new PaddedScanline[ numGlobalBufs];
        for i = 0 to numGlobalBufs-1...
                    globalBufs[ i] = new PaddedScanline(
                      rect.width,
                      rect.left - clipRect.left,
                      clipRect.right - rect.right);

        nextOutLine = 0;
        nextInLine = clipRect.top;
}
```

Figure 31

```
        void renderScanline( LookAroundScanline in, PaddedScanline
        out)
        {
                int downto = nextOutLine;

                // Compute how many scanlines are needed by each
                // object
                for each node in list...
                {
                        node.downTo = downTo;
                        downTo = node.downToNeeded();
                }

                // Empty out the background
                while( nextInLine <= downTo)
                {
                        globalBufs[ nextInLine % numGlobalBufs].fill(
                              RGBA.CLEAR);

                        nextInLine ++;
                }

                // Render necessary scanlines
                for each node in list backwards...
                {
                        while( node.nextOutLine <= node.downTo)
                        {
                                boolean done = node.renderScanline();
                                if( done)
                                {
                                        list.remove( node);
                                        break;
                                }
                        }
                }

                // Composit the result with the background
information.
                in.getScanline( 0).blitTo( out, 0, 0);
                globalBufs[ nextOutLine % numGlobalBufs].
                        mixTo( out, operator, 0, 0);

                nextOutLine ++;
        }
}
```

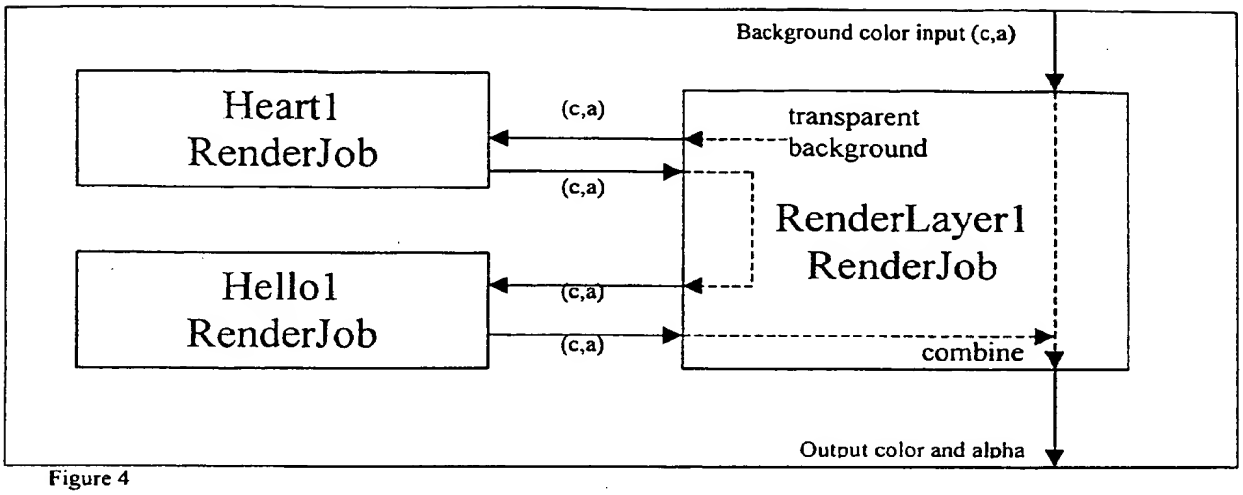



Figure 4

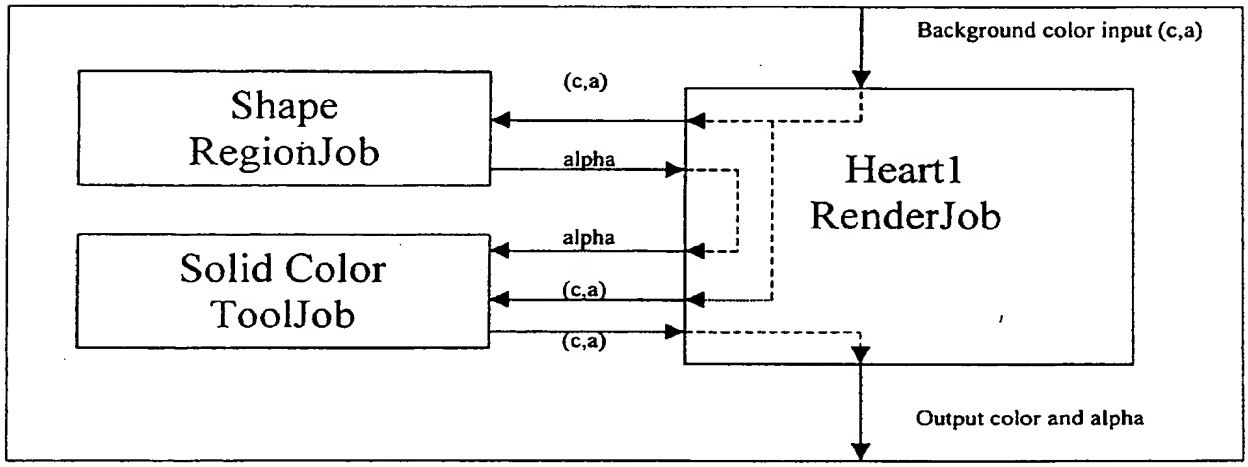
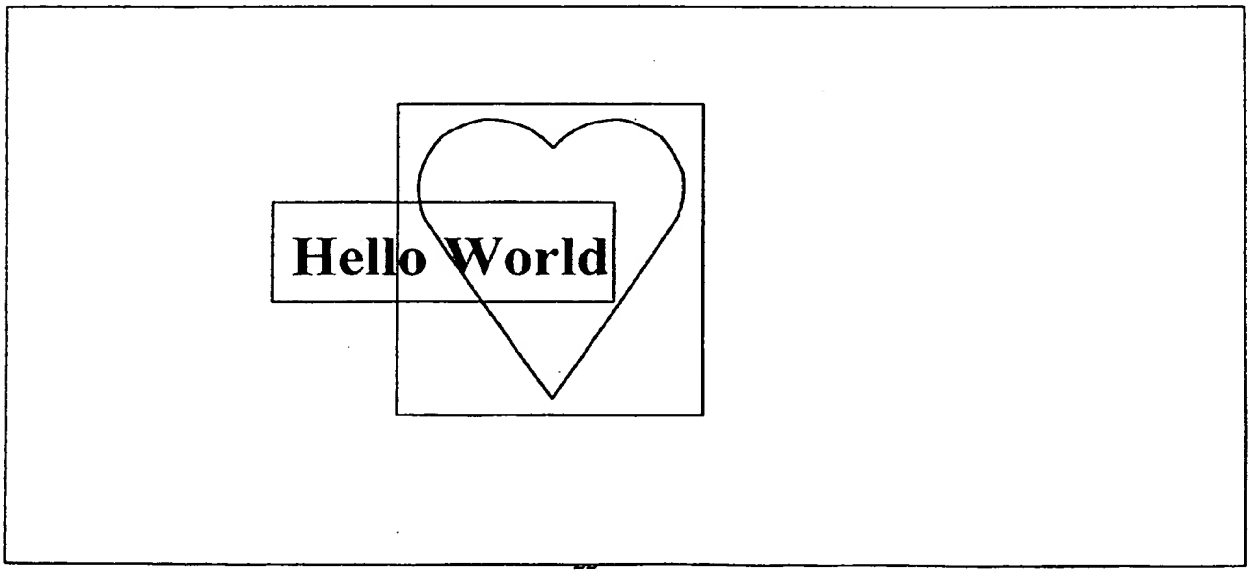



Figure 5 above Figure 6 below



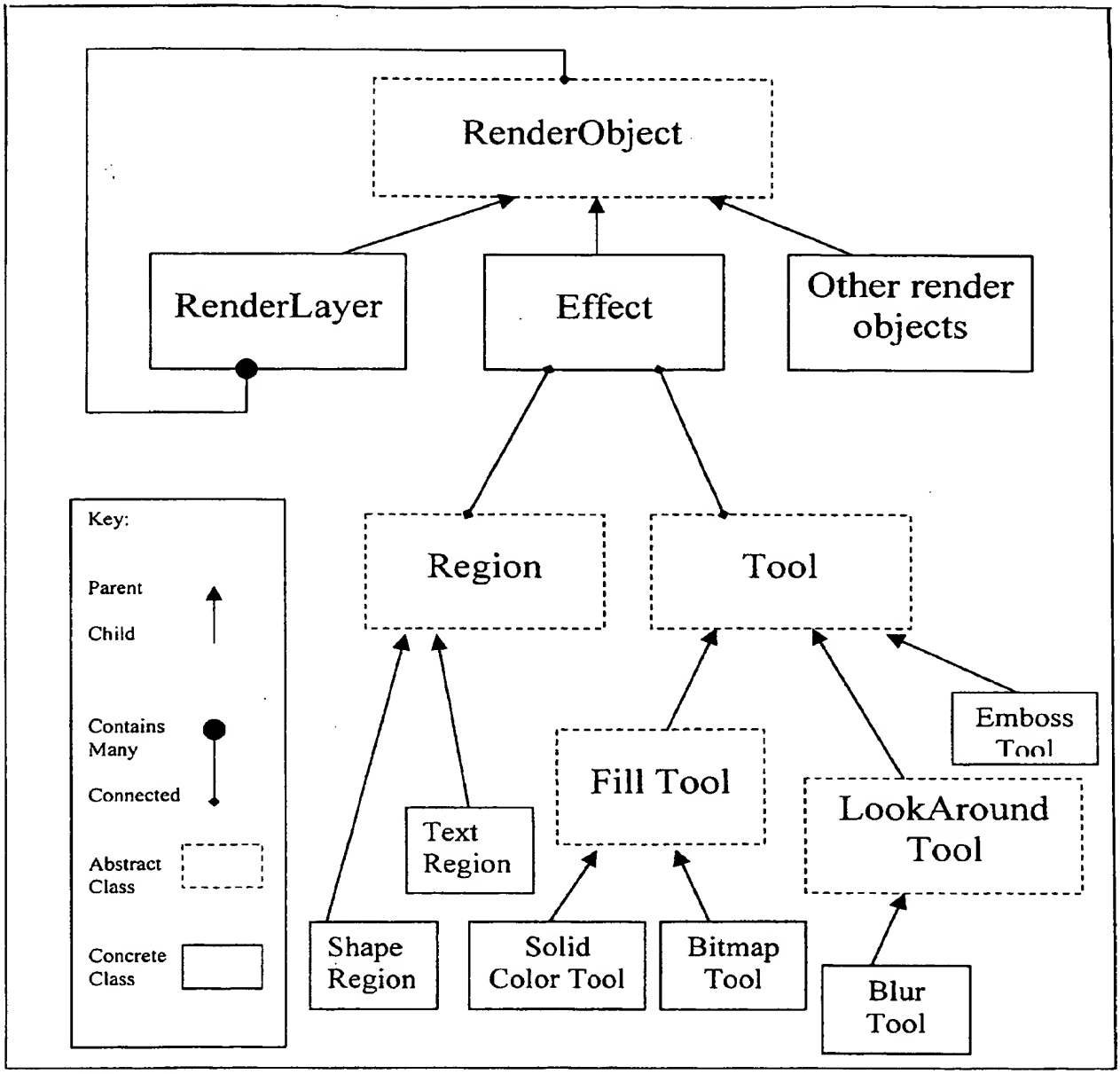



Figure 7 above

Comparison of traditional painter's algorithm and reordered streamed rendering:

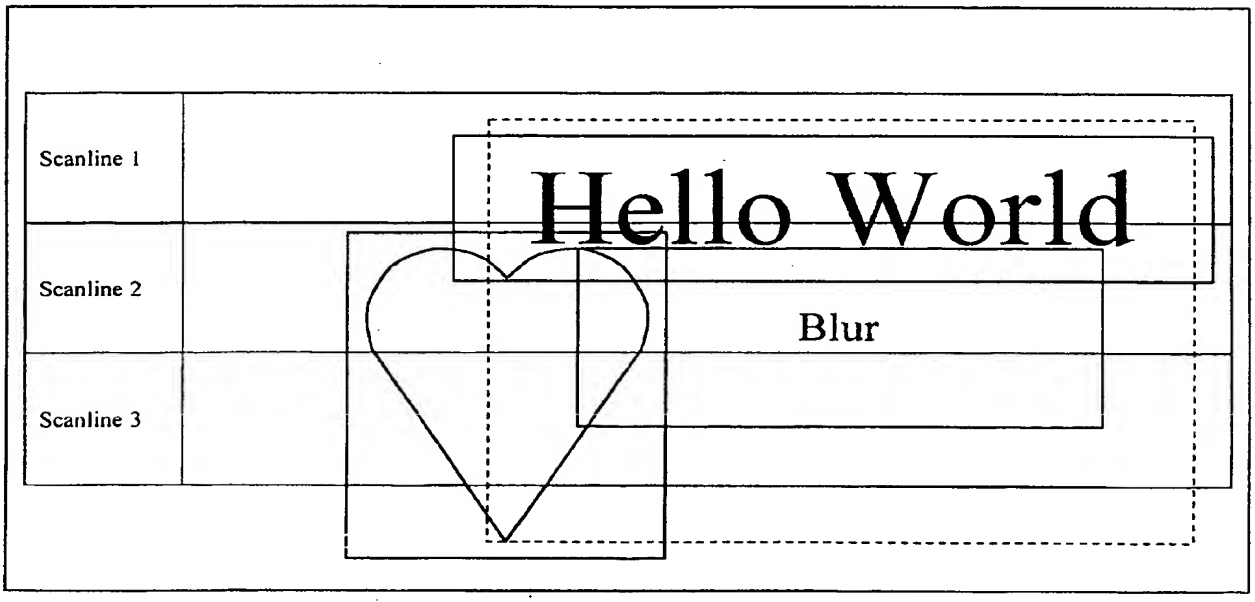
i) Painters

| | Scanline 1 | Scanline 2 | Scanline 3 | (Scanline 4) |
|---|---|---|---|---|
| Output scanline | 12 | 13 | 14 | Inactive |
| Blur1 | Inactive | 10 | 11 | Inactive |
| Hello1 | 8 | 9 | Inactive | Inactive |
| Heart1 | Inactive | 5 | 6 | 7 |
| Background | 1 | 2 | 3 | 4 |

ii) Reordered to minimize buffering

| | Scanline 1 | Scanline 2 | Scanline 3 | (Scanline 4) |
|---|---|---|---|---|
| Output scanline | 3 | 10 | 14 | Inactive |
| Blur1 | Inactive | 9 | 13 | Inactive |
| Hello1 | 2 | 6 | Inactive | Inactive |
| Heart1 | Inactive | 5 | 8 | 12 |
| Background | 1 | 4 | 7 | 11 |

Figure 8 above and Figure 9 below



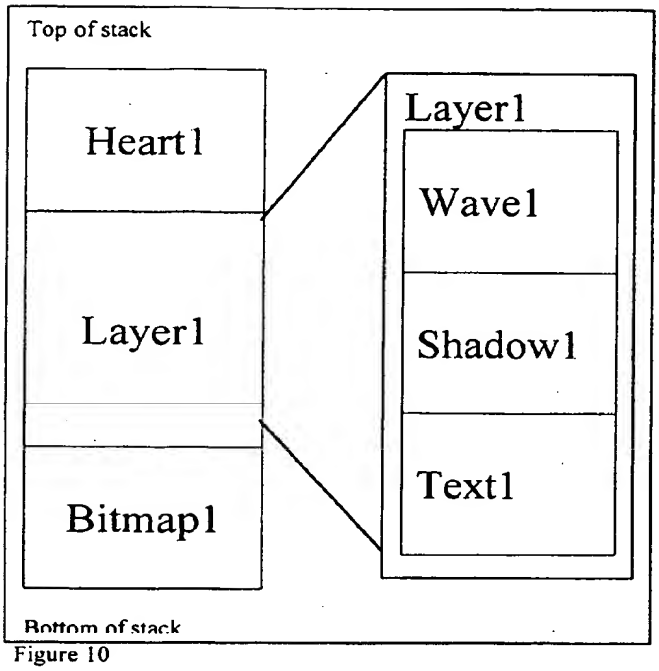



Figure 10

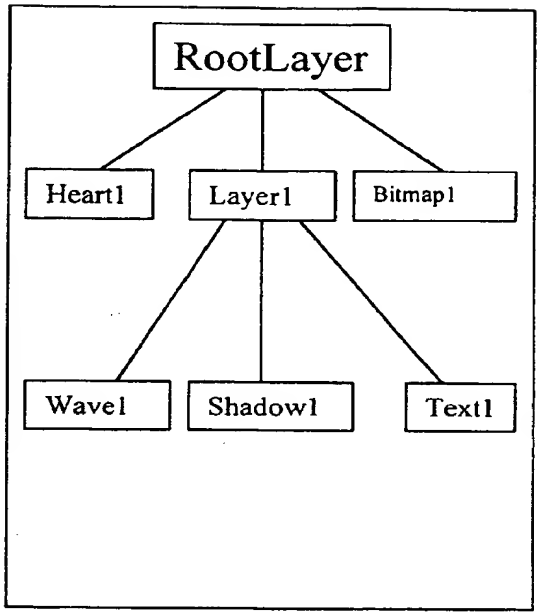



Figure 11


Figure 12